ColdFusion MX Components: A New Methodology for Building Applications

by Benjamin Elmore

April 2002

Copyright © 2002 Macromedia, Inc. All rights reserved.

The information contained in this document represents the current view of Macromedia on the issue discussed as of the date of publication. Because Macromedia must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Macromedia, and Macromedia cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for information purposes only. MACROMEDIA MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Macromedia may have patents, patent applications, trademark, copyright or other intellectual property rights covering the subject matter of this document. Except as expressly provided in any written license agreement from Macromedia, the furnishing of this document does not give you any license to these patents, trademarks, copyrights or other intellectual property.

The Macromedia logo, Flash, ColdFusion, and Macromedia Flash are either trademarks or registered trademarks of Macromedia, Inc. in the United States and/or other countries. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Macromedia, Inc.
600 Townsend Street
San Francisco, CA 94103
415-252-2000

Contents

Comparing ColdFusion MX with ColdFusion 5 1
    Application Structure 1
        Enter CFCs 2
    Team Development 3
Feature Overview of CFCs 4
Basic CFC Development 6
    CFC Terminology 6
    Building a Basic CFC 7
    Working with Components 10
    Calling Components 10
    Packaging CFCs 11
The Methodology of Building a Component-Based Application 12
    CFCs as an Iterative Methodology 12
    Applying CFC Methodology to Applications 14
    Leveraging CFCs in your Development Efforts 14
    Applying the CFC Methodology in Team Development 15
Conclusion 15
About the Author 16

Macromedia ColdFusion MX introduces a new way of structuring your application code into modular units called ColdFusion Components (also called CFCs). More than a way of organizing code, CFCs provide a series of services, such as packaging and class compiling as built-in services that applications can use as a unit. This enables applications to extend ColdFusion functionality to different media, such as Macromedia Flash MX, as web services through Macromedia Flash Remoting, and to other remote services. In some cases, you can only use these new features through CFCs, one of the newest and most powerful features of Macromedia ColdFusion MX.

Comparing ColdFusion MX with ColdFusion 5

Developing applications in ColdFusion MX dramatically differs from developing them in ColdFusion 5. Understanding these differences demonstrates how new ColdFusion MX features will transform your approach to development.

To compare application development in ColdFusion 5 to ColdFusion MX, I will examine two main topics: application structure and team development.

Application structure

To start with application structure, it is necessary to analyze the physical and logical architecture of an application. You can dissect your application into the following sections:

1. Visual
2. Data access/business logic
3. The coordination between 1 and 2

An application that consciously segments its design into these three areas follows a Model View Controller (MVC) architecture (see Figure 1).

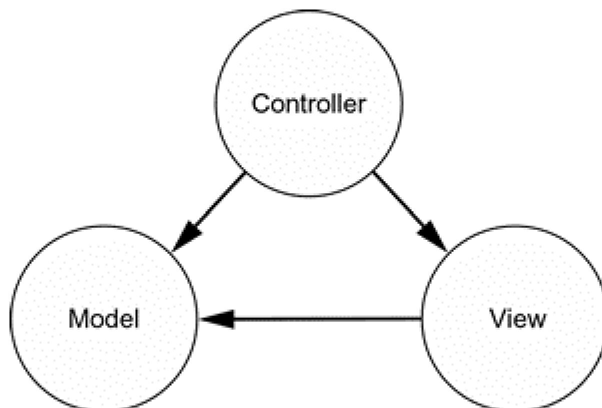

Figure 1: *An MVC architecture*

An application's ability to follow this design depends on the syntax constraints used in the language. For languages that provide no syntax constraints for building an independent module, it is still possible to build an application according to an MVC model by enforcing development practices on a development team. Development practices can enforce an MVC design regardless of language but a language that lends itself to the MVC architecture can make application development easier.

ColdFusion 5 developers can use templates, custom tags and user-defined functions to build applications that follow an MVC architecture. However, one drawback is that the ColdFusion 5 language constructs don't encourage a particular design or approach. Also, without native functionality to encapsulate modules that are independent and grouped, development teams find it hard to follow an MVC architecture.

In other words, the flexibility of ColdFusion 5 (and prior versions) allows developers to create applications in perhaps too many different ways. Although flexibility is usually a benefit in an application, a language with too much flexibility doesn't exactly encourage best practice programming techniques. For instance, while some developers place all logic in the same templates that format the HTML for users, others create applications with multiple tiers of custom tag directories, each of which perform a specific function. Simply put, developers and development teams may organize reusable code based on the context of each application and based on a developer's or development team's preferences, not based on the language's requirements.

Enter ColdFusion Components

By definition, CFCs package related functionality into a single unit. Instead of having multiple custom tags that perform similar functions, which reside in separate files and are organized in a directory structure, developers can organize functionality in a single CFC file. This has a dramatic impact on the CFML language in ColdFusion MX. Coupled with the ability to package components into a larger unit, ColdFusion MX language syntax provides a way to build applications in a MVC model easier. I discuss packaging later in this paper.

Take application development one step further. Each application brings its own set of requirements and, therefore, its own complexity to the mix, no matter what language a developer chooses. However, several application requirements, such as security and syndication of content or functionality, are usually mandatory infrastructure requirements for every application in an organization. The industry has specific standards for building components that fulfill such infrastructure requirements. Two ways to deal with these requirements are either on a case-by-case basis or through a built-in service of a language.

ColdFusion 5 offers no real way of dealing with these common infrastructure requirements, for several reasons. Functionality and data are not grouped in a consistent way (custom tags, UDFs and templates were all options). For example, how does a developer call a web service—a standard for calling remote functionality—if he or she doesn't know where it resides, or in what form it's produced? This means that developers handling infrastructure requirements develop code for each specific case or with specific frameworks built in ColdFusion. Either way, developers add development time to projects or maintenance time to frameworks to maintain them with changing content. In the worst cases, applications may lack some of these base infrastructure requirements due to development time constraints.

Not so in the ColdFusion MX world. One major benefit of CFCs is that they provide a basis in which developers can apply the common infrastructure services to content and functionality. Here is where CFCs dramatically differ from using custom tags. Go back to the example of needing to call functionality for a web service. Because CFCs combine several functions into one unit, they provide an object or class shell from which functions can be called. Also, web services require functionality descriptions (what parameter to pass in, data to output, data types and so forth) that are not possible in ColdFusion 5, but are possible through CFCs in ColdFusion MX. Completing this example, to make a function available to remote services, simply add the attribute with a value of "remote"—such as `access="remote"`—to the `cffunction` tag.

How does this change the infrastructure requirements? For starters, ColdFusion MX applications using CFCs have these infrastructure requirements automatically built in, as well as such industry standards as JAAS and web services. As for development time, developers can spend more time in the planning and designing phase of an application instead of customizing functionality based on an application context.

Team development

Both embedded infrastructure services and a language's syntax constraints strongly impact the way a team develops an application. Project teams can range from one person to a team of twenty or more. Some metrics that monitor how efficiently a development team' uses a language (as a team) include team scalability and team effectiveness.

Team scalability measures whether a team can grow from one developer to twenty and still remain productive. This is important when a project is under a time constraint and the team acquires additional developers. Scalability correlates to the size of a modular unit in a given language. **Team effectiveness** is closely related to a team's scalability but measures a team's ability to use and reuse code and modules from other developers within the same team, or from other teams. It gauges how quickly developers can consume, understand and use that code or module. One major impact on team development that's unrelated to language is the application's architecture and the methodology that is used to build it. Sometimes the language chosen for an application increases the ability of a team to follow a methodology in their application architecture.

Using these metrics to analyze applications built in ColdFusion 5 (and prior versions), you can see that teams achieve scalability through framework choice or a set of strictly maintained best practices by the development team and its managers. Using custom tags and UDF libraries as the basis for code reuse, team scalability depends on the segmentation of design and architecture. Also, these modules lack descriptive information and enforce attributes as built-in features. Because CFCs have built-in metadata and enforced attributes (through the `cfargument` tag), developers can reuse each other's code with ease. Again, this requires that a team properly documents and maintains an interface so that other teams can reuse the module code.

By contrast, CFCs address team development in these areas. They provide features such as packaging for separation, function declaration and validation that are self-documenting. I discuss this further in the next section.

Feature overview of CFCs

ColdFusion MX sets itself apart from ColdFusion 5 because of features made available through components. Below is a list of features that will be covered in the ColdFusion MX Application Developer Center in the CFC Best Practices Series. This month, check out Anthony McClure's article, CFC Best Practices: Component Functions and Function Invocation.

- **Function organization:** This is a central CFC feature. Components are a series of functions (called "methods" when they are associated with a component) that are grouped together into a single unit. Being able to group functions together inside a `cfcomponent` tag allows you to specify a set of other attributes on the `cffunction` tag that describe the function behavior.
- **Property declaration:** In addition to grouping functions together, a component can describe properties that are available to multiple methods (even those in other `cfcomponent` tags). While ColdFusion MX only uses these declared properties when a component is published as a web service, they are exposed through component introspection and become a basis for custom persistence through components.
- **Component inheritance:** This feature allows a component to extend another component. When that happens, the new component definition consists of locally declared methods and properties and those declared in the extending component. This allows a set of base functionality, such as custom persistence, to be reused by multitude components.
- **Custom persistence:** This is an advanced CFC feature that leverages property declarations. Each component method makes a new scope available, called "this," which is readable and editable, and persists for the life of the component. A CFC uses custom persistence when it's built and designed to store the properties of the component in the "this" scope. For architect experts, this means that developers can leverage components in a Value Object pattern.

- **Component class compiling:** This feature distinguishes CFCs from UDFs and custom tags because CFCs are compiled into JavaServer Pages (JSP) page components and then compiled as a Java class with methods without any action from the developer. This provides a dramatic increase in performance.
- **Component packaging:** This feature allows you to place a component into a specific directory and reference it through the directory, in relation to component mapping. Other components in the same directory are packaged together. You can references other components in the same package without the component prefix or their full package name. Also, you can restrict access to component methods or specify which components have access specific component methods inside the same package or subpackages.
- **Web services integration:** Components not only organize functionality, they completely describe the component. To describe methods and functions within a component that are remotely accessible, you can produce a Web Services Description Language (WSDL) file, which makes a component available as a web service. All components can generate a WSDL file by default. You only need to register the web service within the ColdFusion Administrator, which generates the necessary Java Stub components. *For more information on web services, read Stacy Young's article, "Creating Web Services in ColdFusion MX," available on the* ColdFusion MX Application Developer Center**.**
- **Component introspection:** A component description consists of its properties, methods/functions and arguments to its methods/functions. At runtime, you can dynamically determine the component's makeup. The component's properties and methods, and those that it inherits from the other component, are available in the description. You can add attributes to the component creation tags (`cfcomponent`, `cffunction`, `cfargument` and so forth) that are not standard ColdFusion attributes for the tag, but are passed during component introspection.
- **Multiple invocation mechanisms:** When you build an application that supports multiple client interfaces, you must ensure that the client interfaces can each call the component with ease. Interfaces can call components through web services, URLs, the `cfinvoke` tag, and using dot notation (such as `object.method()`) after creating the component through the `createObject()` function. Read more about this in Anthony McClure's CFC Best Practices: Component Functions and Function Invocation.
- **Built-in validation:** When executing a method, this feature validates that the arguments passed to and from the component exist and are valid. It validates both simple and complex data types.
- **Method security:** When describing a component method, you can assign an attribute called "roles" to specify permission for the method. When a page invokes the method through multiple access points, ColdFusion MX verifies that the authenticated user has a valid role assigned to the method.

- **Built-in Component Explorer:** Because components are self-documenting, you can view multiple components located on the ColdFusion server. The Component Explorer displays components that are bundled together into packages, depicts inheritances, shows methods and shows properties. If the component uses the optional `hint` attribute on the component tags (such as the `cfcomponent`, `cfproperty`, `cffunction` and `cfargument` tags), it will provide documentation for the component. You can browse the Component Explorer at this address: http://localhost:8100/CFIDE/componentutils/componentdoc.cfm
Please note that you may need to modify the port or domain in this URL based on your web server setup. You will also need your RDS login and password to access the page.

Basic ColdFusion Component development

Now that you see what components can do, try to build and use one. Although this white paper provides a summary of implementing components, read Anthony McClure's CFC Best Practices: Component Functions and Function Invocation for step-by-step instructions.

Some CFC terminology

CFCs usually consist of three tags: `cfcomponent`, `cffunction` and `cfargument`. You use these tags to create the component logic and group functions as one unit. In more advanced uses of CFCs, developers use the `cfproperty` tag to describe data that's associated with the component. Here is an example of a CFC that uses the basic component tags. The code below is a component named tax.cfc (components always end with the .cfc extension):

```
<cfcomponent hint="A tax component">
 <cffunction name="calcTaxes" access="public" returntype="numeric"
   output="false" hint="This calculates the taxes owed.">
   <cfargument name="taxAmount" type="numeric" required="true"
   hint="The amount to tax.">
   <cfargument name="taxBracket" type="numeric" required="true"
   hint="The tax bracket you are in.">

   <cfreturn arguments.taxAmount * ( arguments.taxBracket / 100 )>

 </cffunction>
</cfcomponent>
```

CFCs introduce some new terminology, as follows:

- **Method:** The distinct set of logic that resides inside of a component. A component can have several methods. This is similar to a UDF, except that methods are always inside of the component; they can only be called if the component is called. The term "method" refers to functions associated with a component in the `cfcomponent` tag. At times, this white paper may refer to functions as standalone user-defined functions. This is an important distinction. The `cffunction` tag defines methods.

- **Argument:** Data element(s) passed to the method in the component. The method can require arguments or specify acceptable data types. The `cfargument` tag defines arguments.
- **Return value:** The return value from a method. Return values are validated against the `returntype` attribute of the `cffunction` tag.
- **Property:** Data element that is associated with the component itself.
- **Invocation:** The process of calling a method within a component.
- **Package:** The location of the component. Component packages are typically created in the webroot or a custom tag path. You can refer to component packages with dot notation, where the dot replaces the slashes in a directory path. (For instance, for a component in com\remotesite\tax.cfc, the dot notation is com.remotesite.tax for the package.)
- **Packaging:** A set of components grouped in the same directory. These components can access other component methods that have an `access="package"` attribute specified in the `cffunction` tag, and a developer can reference any component without its full package name.
- **Tag metadata:** All the component tags (`cfcomponent`, `cffunction`, `cfargument` and `cfproperty`) have attributes that ColdFusion recognizes and uses to produce the desired behavior. It is also possible for a developer to flag additional attributes in these tags as metadata for the tag. Developers can access this information through the `getMetaData()` function and use some of these attributes for custom logic that inspects the components for a behavior or makes the component tags more readable.
- **Inherits:** Components extended from other components. When you describe a component, you specify both methods (through the `cffunction` tag) and properties (through the `cfproperties` tag). If you extend from that component through the `extends` property of the `cfcomponent` tag, the new component inherits the properties and methods of the component from which you extended. Thus both the properties and methods of the parent are available inside of the new component.

Building a basic CFC

ColdFusion components begin with the `cfcomponent` tag and end with the closing `cfcomponent` tag. The `cfcomponent` tag has one explicit attribute, called `extends`. Building on the terminology described in the previous section, you can use the `extends` attribute to describe a parent component so that other components can inherit its properties and methods. To extend this component, you must use the full component's name. Additionally, you can specify a `hint` attribute for the `cfcomponent` tag. While this attribute doesn't provide any functional value, the Component Explorer uses it to describe the component's purpose. The `hint` attribute behaves similarly to the Java Docs in that it describes the component functionality for other developers, so they can easily understand and use your code. As I described earlier, one feature that strongly impacts team development with regards to language is the ability for other developers to consume and reuse code.

Because the tax component example does not extend from another component, you would use the following syntax in tax.cfc:

```
<cfcomponent hint="A tax component">
.
.
.
</cfcomponent>
```

Place the methods and properties for the components within the `cfcomponent` tags. Use the `cffunction` tag to code your method logic. Note that functions describe UDFs in ColdFusion MX and also describe component methods. A function, defined by the `cffunction` tag, is referred to as a method if it's between the `cfcomponent` tags. For step-by-step instructions on how to use a CFC, refer to Anthony McClure's CFC Best Practices: Component Functions and Function Invocation.

The basic composition of the `cffunction` tag includes the following attributes:

- **name** defines the method name that is used when invoking the method. If the component is extending another component and you specify a method name that also exists in the parent component, this method takes precedence.
- **access** determines where the method can execute. There are four access levels: private, package, public and remote. This is one security feature available in CFC functions.
- **roles** validates that the user calling the method is in one of the groups listed in this attribute. This is another security feature available in CFC functions.
- **output** informs the CFC that it can execute its logic, but it cannot produce HTML.
- **returntype** validates the return variable from the method. You set a return value either through the `return` operator in the `cfscript` tag or in the `cfreturn` tag. If the return value does not match the data type specified in the `returntype` attribute, ColdFusion throws an error.

For a method to return a value when called, it must specify a return value in either of two ways. In a `cfscript` tag, use the `Return` statement; in the method, use the `cfreturn` tag. It is not required that you return a value from a method. You can declare a method with no return value (commonly referred to as a subroutine). To do this, do not provide a `returntype` attribute in the `cffunction` tag.

You've added the method and the return value to the component. The method `calcTaxes` is marked as `access="public"` returns a numeric value within tax.cfc. Also note that this example uses the `hint` attribute to document the purpose of the method for the Component Explorer:

```
<cfcomponent hint="A tax component">
 <cffunction name="calcTaxes" access="public" returntype="numeric"
   output="false" hint="This calculates the taxes owed.">
   .
 .
 <cfreturn arguments.taxAmount * ( arguments.taxBracket / 100 )>
 </cffunction>
```

```
</cfcomponent>
```

While passing data out of a method is common, so is passing data to a method with the `cfargument` tag.

The `cfargument` tag typically includes the following attributes:

- **name** specifies the name of the argument to pass to the method.
- **type** specifies the data type passed to the argument. If the passed data type does not match the data type specified in this attribute, ColdFusion throws an exception.
- **required** specifies whether a parameter is required for the method to execute.
- **default** specifies a default value if no argument is passed to the method. If this attribute is present, the required attribute must be set to “no” or not be specified at all.

The `cfargument` tag supports the `hint` attribute, which helps you document the argument usage within the method. In the figure below, if you use the hint attribute in the cffunction and cfcomponent tag, the hints "A tax component" and "This calculates the taxes owed." appear in the Component Explorer (see Figure 2).

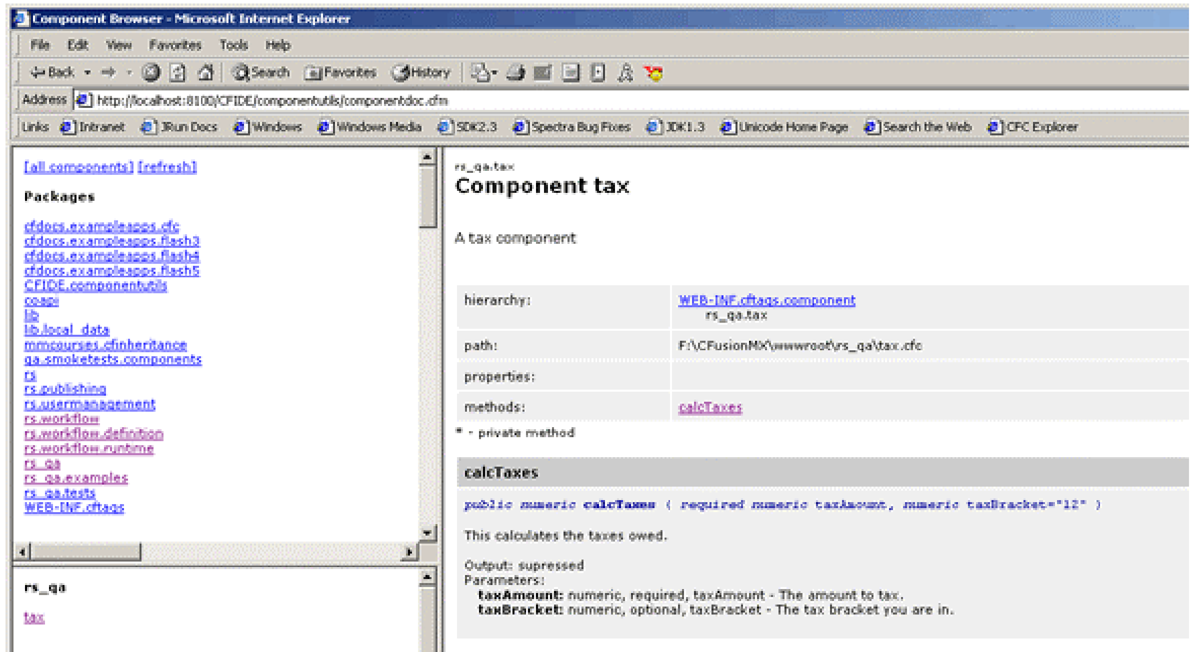

Figure 2: *The Component Explorer showing the `hint` attribute*

Now you have all the necessary background information to build a CFC. The complete tax.cfc component appears as follows:

```
<cfcomponent hint="A tax component">
```

```
    <cffunction name="calcTaxes" access="public" returntype="numeric"
      output="false" hint="This calculates the taxes owed.">
      <cfargument name="taxAmount" type="numeric" required="true"
      hint="The amount to tax.">
      <cfargument name="taxBracket" type="numeric" required="true"
      hint="The tax bracket you are in.">

      <cfreturn arguments.taxAmount * ( arguments.taxBracket / 100 )>

    </cffunction>
  </cfcomponent>
```

When the method is invoked, all arguments passed to the function are stored in the `arguments` scope.

Both methods and arguments can specify a data type for argument validation and return values. Validation choices include any (all), array, Boolean, binary, date, numeric, query, string, structure, object (Java object or component) or a component name. Because components can extend when a component name is provided, the argument validates the data to see whether the component passed is of the component type or any component that extends from it. This validation also provides a way for you to self-document code in the Component Explorer.

Working with components

CFCs extend the functionality of your ColdFusion MX applications through web services and a packaging model.

Calling components as web services

CFCs provide many features that set it apart from developing applications in ColdFusion 5 (and prior versions). Available methods and arguments are descriptive and accessible to remote services. This gives you the ability to make a component available as a web service, which requires a firm definition for its methods.

The order in which the arguments appear is important. ColdFusion MX introduces several ways for accessing a component as web services: from a Macromedia Flash client, from a URL called from another ColdFusion page, from another component and from the `cfinvoke` tag.

Accessing a component as a web service or from a Macromedia Flash client requires that the function include a set method signature. A set method signature is the method name plus its arguments. For example, the method signature for `calcTaxes` is `calcTaxes(taxAmount, taxBracket)`. The way that CFCs generate the ordering for the arguments that a set signature needs is by looking at the order in which the `cfargument` tags are placed in the code. The cool thing here is that by using optional arguments, you can create multiple signatures for the same method. Optional arguments are defaulted if they're not provided so a method can be called successfully without any of its optional arguments. It's in this way that you can get multiple signatures from one method. In the tax.cfc example, `taxBracket` is optional, so another signature to the method would be `calcTaxes(taxAmount)`.

Being able to invoke components as web services, and through Macromedia Flash Remoting, provides you with the ability to design applications that accommodate different client interfaces. Calling a set of modular logic no longer requires a request to a ColdFusion page and then to your logic. In essence, the availability of a built-in infrastructure service makes your application accessible immediately through multiple clients, without writing any additional code. In fact, to make a component a web service, just add the attribute `access="remote"` to the `cffunction` tag and register it in the ColdFusion Administrator under Web Services.

Packaging CFCs

Where you save your file is very important because it's where the component gets its name. Packaging is more than just putting components with similar functions into the same directory with a coherent directory structure; it also means that the components can import and use methods from other components. This feature allows the components to work as a group. Components outside the packages cannot have access to the package-accessible methods, which further tightens the modularity of packages.

Here's how you refer to a package name. Start with the component name, the CFC filename, without the .cfc extension (so it would be "tax" for "tax.cfc"). Place its package name in front of the filename. The package name describes the directory.subdirectory structure from either the webroot or a custom tag path to the component file. So if the tax component were located at *wwwroot*\com\remotesite\tax.cfm, the package name would be 'com.remotesite.tax'. As you'll see later, grouping these components into a package provides the perfect unit to build CFC architecture.

Methodology of building a component-based application

A methodology is the process that a developer or development team follows to develop applications with efficient use of team resources and strong, flexible architecture. Because applications are vast in their requirements and scope, it's impossible to develop one methodology that fits all scenarios. Instead, you'll find that several methodologies tailor themselves to a variety of different objectives, such as a short implementation timeline, a language, light business requirements and so forth. When an application's scope is evident, it's easier to compare methodologies that are appropriate and those that are not.

While it's possible to create a methodology that assists developers using a particular language, it's not very common. Instead, methodologies leverage different languages indirectly through the language's approach. For example, a methodology that moves with the development lifecycle—one that gathers all requirements first and then goes to development (using a **waterfall methodology** or model)—is better suited for procedural languages than a methodology that is highly iterative (such as an **iterative methodology**). These methodologies are defined below.

Let's look at the **waterfall methodology** as an example. To back up for a moment, WhatIs.com defines the waterfall methodology as a "development method that is linear and sequential. Waterfall development has distinct goals for each phase of development. Imagine a waterfall on the cliff of a steep mountain. Once the water has flowed over the edge of the cliff and has begun its journey down the side of the mountain, it cannot turn back. It is the same with waterfall development. Once a phase of development is completed, the development proceeds to the next phase and there is no turning back." The waterfall methodology doesn't make provisions that are language-specific. Instead, it contains both team development and architecture approaches towards building your applications. Looking at these approaches you can evaluate a language and then evaluate if it can be efficiently used in that methodology. The key there is "efficiently used" because we all know we can force anything.

CFCs as an iterative methodology

A development team can use a procedural language as an **iterative methodology**, but not nearly as effectively as an object-based or object-oriented language can. A strong example of a procedural language is ColdFusion.

Whatis.com defines an iterative methodology as an approach where a development team develops an application "in small sections called iterations. Each iteration is reviewed and critiqued by the software team and potential end-users; insights gained from the critique of an iteration are used to determine the next step in development. Data models or sequence diagrams, which are often used to map out iterations, keep track of what has been tried, approved, or discarded, and eventually serve as a kind of blueprint for the final product. "

"The challenge in iterative development is to make sure all the iterations are compatible. As each new iteration is approved, developers may employ a technique known as backwards engineering, which is a systematic review and check procedure to make sure each new iteration is compatible with previous ones. The advantage of using iterative development is that the end-user is involved in the development process. Instead of waiting until the application is a final product, when it may not be possible to make changes easily, problems are identified and solved at each stage of development. Iterative development is sometimes called circular or evolutionary development. "

With ColdFusion (ColdFusion used outside of a framework), an iterative approach such as the Rational Unified Process (an iterative methodology created by Rational Software) requires a massive coordination effort by the development manager(s), technical manager(s) and architect(s).

So far, this article has discussed the features of CFCs in ColdFusion MX, the differences in developing applications in ColdFusion 5 and ColdFusion MX, and CFC fundamentals. It would be safe to say that development teams use methodologies differently in building ColdFusion MX applications, or at least use methodologies more efficiently.

To learn more about applying a structured approach to methodology, read my articles, Introduction to an Integrated Development Methodology, Part 1 and Introduction to an Integrated Development Methodology, Part 2. These articles introduce the “integrated development methodology” and detail the typical goals and objectives of methodologies.

After reviewing this methodology, you’ll see it has a set of approaches for both team development and application architecture. Methodologies don’t necessarily specify a language with which to build an application. In analyzing the methodologies, you can see that some languages are better suited for methodologies than others. In an integrated development methodology, the main theme is iterative development against core modules or packages of functionality. In short, a language that has a concept of a unit or a way to package units of functionality together can leverage this methodology effectively. ColdFusion Components are just that feature.

In addition, a methodology exposes the strengths and weaknesses of a language. CFCs are a strong language feature in ColdFusion MX that include packaging, inheritance, introspection (reflection), polymorphism, method security and custom persistence. From these features, a development team can build according to different designs or well-known patterns (reusable approaches to problems).

However, CFCs do have some weaknesses. These include the inability to call an overloaded method on a parent object, invocation introspection, and interfaces. Architecture and designs patterns are huge topics, and books have been written about them; they can’t fit into this white paper. An important point to take away, however, is to realize that languages can help or hinder design patterns within application development. These are fundamental considerations in implementing your design.

Applying CFC methodology to applications

Generally, it's much easier to dissect an application from the top-level and work into the nested layers. For an e-commerce application, for instance, you would start breaking down the higher-level functionality such as Customer Management, Products Display and Ordering. To dissect the application functionality further, a development team might ask what supports these three visible functional areas of the application. They would continue to break down the functionality into smaller and smaller chunks.

This functionality breakdown is the basis for development iterations and team development. The development team looks at each specific section and analyzes its functionality to see what data is persisted and how many different activities it should perform. From this analysis, the team determines a few things: which CFCs store the data (customer, product and so forth); which components are interactive (such as the shopping cart); and what needs do the components share? In one possible scenario, you would need to persist multiple content types and ask whether you should provide a generic content storage CFC to make the work easier. It is from this step that the development team identifies the infrastructure CFCs for the application.

Hold on before you apply developer resources. The next section takes this functional breakdown and applies it to team development.

Leveraging CFCs in your development efforts

ColdFusion Components positively impact the capabilities of team development. At the beginning of this paper, I described metrics that an application development team uses to measure its scalability and effectiveness. (Scalability is whether a team can grow from one developer to twenty and still remain productive; team effectiveness determines how a team is able to reuse code or modules from other developers or teams.) CFCs have a strong set of features that increase team scalability and effectiveness.

When a project requires, due to time constraints, that you add developers to your team, it's important that the language allows developers to segment functionality easily and that the methodology allows for multiple teams to develop code in iterations and fit them together. The methodology earlier described an iterative process and how to breakdown an application's functionality.

Focusing on the language requirements, look at segmentation in two areas: tiers and functionality. Application tier segmentation is the separation of all functionality into layers such as View, Process, Data, Control, and so forth. The opening example shows that Model View Control (MVC) architecture is an application-tier segmentation. The second type of segmentation is based on the functionality of the application. Inside each tier, you can separate into grouping functional requirements. For example, your application may involve a workflow subsystem and customer management subsystem. Both of these subsystems could be in the same tier; but from a functionality standpoint, they are separate (even if one leverages the other). CFCs handle segmentation at both levels.

Because CFCs can expose certain methods as `remote`, CFCs achieve application-tier separation. When a CFC method is `remote` it can be leveraged by both a Macromedia Flash component and by remote web services. This enables you to write tiered applications independently without knowing how they will be consumed. With CFCs you can expose methods within your application to remote services as web services. You would only need to set access=remote in your cffunction tag, and this would turn the method function into a web service that allows other ColdFusion Pages, Macromedia Flash components (through Flash Remoting), or other remote services to use it. For instance, you might want a remote entity to participate with a workflow subsystem. With CFCs this is effortless.

In truth, without an architecture or design that supports separation into tiers (such as MVC), this feature or any feature of CFCs becomes invalid. CFCs scale well in the application tier. Once inside an application tier, it's best to separate according to distinct functionality. Functionality segmentation flows around two types of bundling that CFCs provide: the component itself and a package of components. For instance, a development manager might assign one developer per component, one developer per package or a team of developers on a specific package.

Applying the CFC methodology in team development

You've had a glimpse of how CFCs and packaging affect team development. For instance, there are teams that work against the client, the middleware and in the data tier of applications. After teams are assigned to the appropriate tier, they analyze functionality segmentation and assign subteams and specific developers for each set of functionality. Larger teams may assign members to work on more granular pieces, whereas smaller teams may work on complete functionality.

With the possibility of either a granular deconstruction or a high-level breakdown of the application, the last item I discuss is the impact of language and methodology on team productivity as team resources increase. For instance, a small development team is asked to update code built by another team. The common problem is that the new development team must try to understand the framework with which the original team created the application. Although inline comments are usually sufficient (when a development contact is still available), it's usually more common for the code comments to become out of sync with the code.

CFCs solve this problem. First, developers can add the optional `hint` attribute to the `cfcomponent` tag, which shows up in the Component Explorer. Second, as a developer updates a method with new arguments, the documentation for the component method updates as well. This self-description allows developers to understand and rapidly consume and use components from other developers. Lastly, developers can use the `cfargument` tag in a `cffunction` tag to validate parameters passed to the method. These arguments also allow for arguments of a specific type to be checked.

Conclusion

With the introduction of CFCs in ColdFusion MX, it's clear that there are new options for application development. This strongly impacts team development and application structuring. CFCs are flexible enough to form the core of an architecture or simply offer built-in functionality, such as producing and consuming web services. Web service generation, component packaging and code self-documentation give developers the ability to create flexible, well-architected, well-documented applications without adversely impacting development cycles.

For more information on developing ColdFusion components, read more in the ColdFusion MX Application Developer Center.

About the author

Benjamin Elmore is Chief Technology Officer at RemoteSite Technologies Inc.*, an Internet consulting and training company focused exclusively on enabling clients, students, businesses and employees to attain success with advanced web technologies. A well-known member of the community, Ben is on the board of advisors for the Albany ColdFusion User Group. He also gives free talks at user groups across the nation and presents topics at various international developer conferences, such as those for Rational Software, Macromedia, and Sybase. Recently, this certified Macromedia ColdFusion instructor gave presentations at the Macromedia Worldwide Developer's conference. Ben coauthored* Macromedia Advanced ColdFusion 5.0 *with ColdFusion evangelist Ben Forta and has written numerous articles for the* ColdFusion Developers' Journal *and numerous international periodicals. Currently, Ben is a consultant on the development team redesigning* macromedia.com*, as it integrates former allaire.com content using the Macromedia MX strategy.*